A graphical processing unit accelerated NORmal To Anything algorithm for high dimensional multivariate simulation

Xiang Li[†], A. Grant Schissler[‡], Rui Wu[§], Lee Barford[†◇], Frederick C Harris, Jr.[†]

[†]*Department of Computer Science and Engineering*
University of Nevada, Reno
Reno, NV

[‡]*Department of Mathematics and Statistics*
University of Nevada, Reno
Reno, NV

[§]*Department of Computer Science*
East Carolina University
Greenville, NC

[◇]*Keysight Laboratories*
Keysight Technologies
Reno, NV

xli@nevada.unr.edu, aschissler@unr.edu, wur18@ecu.edu, lee.barford@ieee.org, fred.harris@cse.unr.edu

***Abstract*—Many complex real world systems can be represented as correlated high dimensional vectors (up to 20,501 in this paper). While univariate analysis is simpler, it does not account for correlations between variables. This omission often misleads researchers by producing results based on unrealistic assumptions. As the generation of large correlated data sets is time consuming and resource heavy, we propose a graphical processing unit (GPU) accelerated version of the established NORmal To Anything (NORTA) algorithm. NORTA involves many independent and parallelizeable operations — sparking our interest to deploy a Compute Unified Device Architecture (CUDA) implementation for use on Nvidia GPUs. NORTA begins by simulating independent standard normal vectors and transforms them into correlated vectors with arbitrary marginal distributions (heterogenous random variables). In our benchmark studies using a Tesla Nvidia card, the speedup obtained over a sequential NORTA coded in R (R-NORTA) peaks at 19.6x for 2000 simulated random vectors with dimension 5000. Moreover, the speedup obtained for GPU-NORTA over a commonly used R package for multivariate simulation (the COPULA package) was 2093x for 2000 simulated random vectors with dimension 20,501. Our study serves as a preliminary proof of concept with opportunities for further optimization, implementation, and additional features.**

***Index Terms*—GPU, parallel computing, CUDA, multivariate statistics, multivariate analysis, Bioinformatics, Monte Carlo, NORTA, simulation**

I. Introduction

Many stochastic simulations cannot be accurately represented by independent and identically distributed samples. Realistic simulations must include correlations and allow for heterogeneous marginal distributions, modeled using a correlation matrix. For example, Biological processes are often highly correlated, involving many complex interactions. Applications in integrative analysis of multiple data sets has been recently researched [1]. One of the pitfalls of not accounting for correlations was researched by Gatti et al. [2]. They found significantly higher rates of false positives in gene set testing when independent genes were inappropriately assumed. In order to accurately represent this data, a multivariate (correlated) structure must be used. The expression of genes is also heterogeneous, that is not all genes follow the same distribution. Using the NORTA method described in this paper, we are able to simulate random vectors whose components follow arbitrary distributions marginally while jointly reflecting a specified correlation matrix.

The ability to simulate high dimensional correlated data has applications in machine learning as well. For example, Wilkins, Morris, and Boddy researched methods of using backpropogation neural networks to identify marine phytoplankton using multivariate flow cytometry data [3]. In identification problems, lower dimensionality leads to increased overlap between classes. For higher accuracy, it becomes necessary to use the full multivariate nature of the data to distinguish between classes. Another common simulation method is the Monte Carlo Method which was used recently by Russkova to model weather phenomena [4].

There are numerous uses of multivariate statistical analysis, ranging from RNA sequence analysis to health records [5]. More recently Often complex real world system are composed of dependent variables that cannot be treated as independent. The use cases are extremely broad and as computing power continues to increase, use of high performance algorithms for preforming these statistical analyses will become more and more commonplace.

In general, methods for generating correlated random vectors can be classified into three categories [6]:

1) Analytic methods using conditional distributions.
2) Numerical approaches including accept/reject methods.
3) Transformation of univariate vectors.

Numerous approaches researched in the first two categories are limited to bivariate distributions and can only be used for generation of variables that share a common distribution [6].

NORTA falls into category 3. The advantage of employing this method is in its simplicity, propensity for parallelization, and broad applications. Algorithms in this category are able to use the appropriate set of marginal probability distributions and a correlation matrix at the cost of partially specifying the joint distributions [6]. This technique offers the benefit of being general, while avoiding solving complex equations [7]. The NORTA method transforms elements from a multivariate standard normal distribution to the desired marginal distributions [6]. A more detailed description of the algorithm is given in Section III.

For large and dense matrices, the number of operations scale with the number of elements, potentially leading to compute time bottlenecks. Analysis of the bottlenecks of a sequential implementation is discussed in Section IV. Therefore, implementation through parallel computing algorithms on the GPU becomes attractive. Due to the Single Instruction Multiple Data (SIMD) architecture of GPUs, operations can be performed in parallel thus providing a speedup over a sequential implementation.

The rest of the paper is organized as follows. Section II provides a brief summary of statistical background relevant to the main algorithm used, as well as an introduction to the GPU architecture, followed finally by a discussion of unified memory. Section III discusses the NORTA algorithm implemented in this paper. Section IV reviews the sequential implementation written in R. Section V reviews the GPU implementation. Section VI includes the testing methodology and experimental results followed by a discussion in Section VII. Section VIII presents the conclusion and future work.

II. Background

A. Statistical Background

The first topic to discuss is the correlation matrix. This matrix is positive semi-definite symmetric matrix with dimensions $d \times d$, where d corresponds to the number of variables in a multivariate vector whose correlations are represented by the matrix. Each entry in the matrix, with indices (i, j), represents the Pearson correlation coefficient $\rho(X_i, X_j)$ between random variables X_i, X_j given by the equation [8]:

$$\rho_{X_i,X_j} = \frac{E[(X_i - \mu_{X_i})(X_j - \mu_{X_j})]}{\sigma_{X_i}\sigma_{X_j}} \tag{1}$$

E is the expected value. μ_{X_i}, μ_{X_j} are the means and $\sigma_{X_i}, \sigma_{X_j}$ are the standard deviations of X_i, X_j respectively.

The matrix is symmetric because correlation between X_i and X_j is the same as the correlation between X_j and X_i. Also the main diagonal is composed of d 1s, reflecting the fact that a random variable is perfectly correlated with itself. Essentially, a correlation matrix represents the relationship between variables in a multivariate vector.

The Cholesky decomposition is used because it is an efficient method for decomposition of an input correlation matrix (and, in general any semi-positive definite matrix) [9]. The Cholesky decomposition provides two factors: one lower and one upper triangular $n \times n$ matrix. Multiplication using one of these factors onto a matrix of independent random variables will induce a specified correlation [10].

For convenience and ease of simulation, the NORTA algorithm begins with independent and identically distributed normal vectors. Most pseudo-random number generation software include the ability to simulate random normal variables. Further, we choose to start with normal vectors to begin our stochastic simulation to align with the original NORTA algorithm. One could start from other distributions, such as marginally uniform random variables. The important part is that we begin with any suitable random variable that can be easily transformed into a copula by applying the distribution's inverse cumulative distribution function (CDF). A copula is a multivariate distribution with uniform marginals and an arbitrary correlation matrix [11]. Importantly, Sklar's theorem [12] shows that any joint distribution has copula representation — providing a guarantee for the success of the NORTA algorithm.

The inverse (probability) transform is a general method of transforming one random variable into another. The transformation begins with values (usually a vector) obtained from one distribution with known CDF. Then the known CDFs are applied to these values to obtain values that have a uniform (on the interval 0 to 1) distribution. These values can be thought of as probabilities that a value from the original distribution was less than or equal to the original, observed value. Then these probabilities can be transformed into another target distribution by applying the inverse CDF of the target distribution. See Rizzo 2007 for details [10].

B. GPU architecture

GPUs feature single instruction stream multiple data stream architecture that is optimized for data-parallel computations. This architecture works well for applications running a single instruction set over many different data elements. Compared to traditional central processing unit (CPU) architectures, the GPU is throughput focused. A GPU accomplishes this by having many compute cores that are able to process thousands of calculations simultaneously. While a GPU is conventionally used for graphics displays, in the past decade or so, there has been a large growth in general purpose GPU computing (GPGPU) [13]. Nvidia has created their own API for GPGPU programming using their chips. This language/API is an extension of the ANSI C language and is called CUDA which stands for the Compute Unified Device Architecture. Functions that are to be ran on the device (GPU) are called kernels. Each thread launches an instance of the kernel to run.

Kernels on the graphics processor is organized into grids of blocks of threads. Figure 1 visualizes this structure. Threads in the same block are run on the same streaming multiprocessor, which is the Nvidia compute unit. The advantage of using a GPU in parallel computing and high performance computing lies in the fact that Nvidia GPU's are commodity hardware and readily available for purchase. They are also fairly inexpensive and easy to setup compared to the more traditional computer cluster that are often used in high performance computing

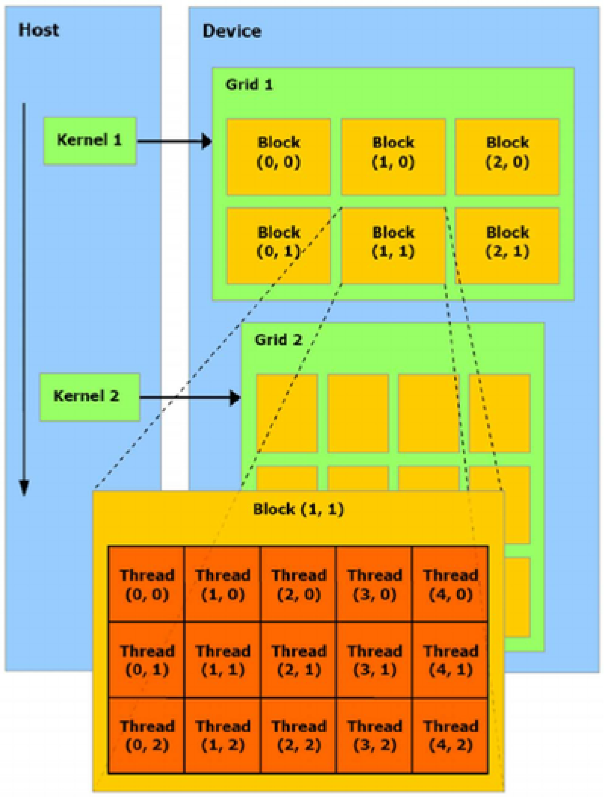

Fig. 1. Nvidia thread block structure: Each kernel corresponds to a grid. Each parallel invocation of the kernel corresponds to a block, and each block can be further divided into threads. [13], [14].

(HPC). With a GPU, high performance computing becomes more accessible, being able to speedup scientific computation by a significant amount without having the cost of setting up a CPU cluster. The GPUs used in this work were a Tesla P100 and an Nvidia GTX 1080 which are part of the Pascal architecture family.

C. *Unified Memory*

CUDA tool kit 6 introduced unified memory to the CUDA ecosystem [15]. CUDA programming up until this point required manual memory manipulation between the device and the host (CPU). With unified memory, the programmer no longer has to explicitly move data between memories. Memory management is handled by the CUDA backend and is abstracted from the programmer. This is illustrated in Figure 2.

The Pascal architecture, which is the architecture of GPUs used for this paper, leverages a built-in hardware Page Migration Engine to handle page faults and data migration [16]. Unified memory is especially useful when the data set is too large to fit on the device. Without unified memory, the developer would have to manually chunk their data and transfer it between CPU and GPU. In the case of this project, the largest input matrices are about 3.4 GB for double precision numbers. This is quite large and can pose memory issues when used with common GPUs with 4 GB and less memory, especially with larger values of simulation replicates, n. Although unified

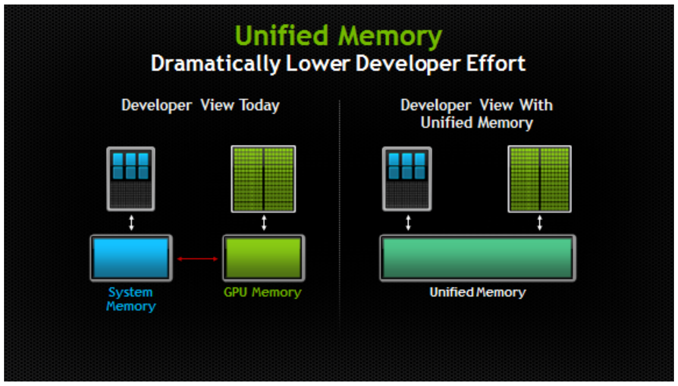

Fig. 2. Nvidia's Unified Memory combines physically distinct CPU and GPU memory into a single memory address space and handles data migration between the two automatically [15].

memory reduces the complexity of CUDA code, it is relatively new and not optimized yet. Therefore, it is reasonable to speculate that the algorithm could run faster if using manual data migration.

III. NORTA ALGORITHM

Algorithm 1 describes the NORTA algorithm for the generation of a $d \times 1$ random vector X. This algorithm is extended to an $d \times n$ matrix, where n is the desired number of simulation replicates obtained by repeating the procedure. For a more in-depth explanation of the NORTA algorithm, refer to Cario [7].

The NORTA algorithm is divided into four steps (lines 2-5). The first step (line 2) is the Cholesky decomposition of the input correlation matrix, Σ_z, into a product of an upper triangular and it's conjugate, M and $M^{'}$ respectively. The input symmetric matrix has dimension $d \times d$. This step is the precursor to inducing the specified correlation matrix. The next step (line 3) is to generate a $d \times 1$ independent and identically distributed normal vector W. Then (line 4), W is applied the proper correlations using a matrix multiplication between M and W to return Z, a multivariate normal vector. In the last step (line 5), each element of Z is applied $\Phi(x)$, which is the standard normal CDF. Then, the inverse CDF of desired marginal distribution, also known as the quantile function, $F_{X_i}^{-1}(x)$ is applied. This will return, X, the final transformed vector, which is repeated n times to get the simulation matrix.

Algorithm 1 NORTA algorithm

```
1: procedure NORTA
2:     Produce $M$ of $\Sigma_z$ so that $MM' = \Sigma_z$.
3:     Generate $W = (W_1, W_2, ..., W_d)^{'} \leftarrow d \times 1\ vector$.
4:     Set $Z$ by $Z \leftarrow MW$.
5:     Return $X$ where $X_i \leftarrow F_{X_i}^{-1}[\Phi(Z_i)],\ i = 1, 2, ..., d$.
```

Importantly, we deviate from the original NORTA algorithm in that we do not adjust the input correlation matrix to get a (near) exact target correlation matrix from our final simulation data set. Instead, we use the target correlation as input and forgo the massive, brute force search for an adjusted input correlation matrix suggested by Cairo and Nelson [7]. Later, we assess the loss of accuracy associated with using the target correlation matrix by comparing to the publicly available COPULA R package [17].

IV. Sequential Implementation

The sequential version of this simulation program was implemented using the R programming language. R is commonly used by statisticians and has many built-in statistical functions. This made the implementation of the inverse transformation much easier as R has support and optimization for all of the distributions used for this project are available. The random number generator, quantile, and distribution functions of each type of distribution were used for the sequential implementation. In addition to statistical functions, R also features support for working with dense and sparse matrix math. The Cholesky decomposition, matrix multiplication and random matrix generation were also handled through built-in functions. A text file containing the correlation matrix and a text file specifying marginal distributions with parameters were the inputs to the program. The output is a text file containing the final transformed simulation matrix.

The Cholesky decomposition, random number generation, matrix multiplication, and inverse transformation steps of the algorithm were expected to benefit the most from GPU parallelization because, due to the number of independent operations, the computation time scales with size of matrix for sequential implementations. Results from bottleneck analysis in Table I confirms this. Analysis of the inverse transformation step was not included because it was not able to be parallelized in this study due to difficulties explained in Section V.

V. GPU Parallel Implementation

The parallel GPU version of the NORTA algorithm was implemented using the CUDA programming language. Input and output data were the same as in the sequential implementation described above. The `matrix_read_in` function and `distribution_list_read_in` functions were implemented sequentially in the GPU version of the code. Memory management was handled automatically by CUDA unified memory.

Several official CUDA APIs from Nvidia were used. They were: `CuBLAS`, `CuRAND`, and `CuSOLVER` [18]–[20]. Library functions abstracted out the notion of kernels. The Cholesky decomposition was implemented using the `potrf` function from CuSOLVER. This function performs the Cholesky factorization and returns an upper or lower triangle in column major order depending on arguments specified and uses double precision. Matrix multiplication was implemented using the `gemm` function from CuBLAS. `gemm` performs matrix multiplication given two input matrices and stores them in an output matrix specified in the arguments. The output matrix used was one of the original input matrices for better memory efficiency. Double precision was used for this function as well. `CuRAND` was used for the generation of the random normal matrix. The pseudo random number generator was used out of this library using the default generator to generate random numbers in double precision sampled from a standard normal distribution.

For the inverse transformation function, the C++ stats library `StatsLib` made by Keith O'Hara was used [21]. Distributions supported are: Beta, Cauchy, Exponential, F, Normal, Log Normal, Logistic, Poisson, t, Uniform, and Weibull. This library was chosen because it was one of few C++ statistical libraries found that imitated the R distributions library. Implementing distribution functions is a complex process so due to time and technical ability constraints, distributions were not written directly in CUDA. For testing purposes, only the Poisson distribution was used. The reasoning was that the input data came from RNA sequence data may be modeled by Poisson. Due to difficulties stemming from incompatibilities between C++ and CUDA, the inverse transformation function was also implemented on the CPU.

VI. Experiments and Results

The CUDA libraries and functions used in this study treat matrices as column major instead of the C style row major. This causes most functions such as the Cholesky decomposition and the matrix multiplication routine to return the transpose of the expected result. This implicit transpose is caused by the conversion from row major to column major. Care must be taken to get the correct results out of these routines. Unified memory was used to store all arrays. This allows arrays to be larger than the memory of the GPU. Timings were captured for various n and d values. Speedup was calculated between GPU version and sequential version. Speedup vs another popular R library for multivariate statistical simulation called COPULA was also captured. Refer to Yan for details of the COPULA R package [17].

Once the implementations discussed in Section IV and Section V were finished, timings for the various modules of each implementation were done in order to see how the R and CUDA computations compared. Timings for the sequential R column shown in Table I were done on a Macbook Air with an Intel Core i5 processor at 1.3 Ghz and 4 gb of memory. The CUDA column in Table I was performed on a CUBIX box with dual 12 core Intel Xeon CPUs running at 2.00 GHz. This machine has 8 Nvidia GeForce GTX 1080 GPUs but only one was used in the computation of this project. A different machine was used than the timing data collected for sequential R column due to the fact that the Macbook Air used did not have a CUDA compatible graphics card.

TABLE I
TIMINGS COMPARING SEQUENTIAL R AND PARALLEL CUDA IMPLEMENTATIONS FOR $d = 20501$, $n = 1094$ (IN SECONDS)

Functions	**R (s)**	**CUDA (s)**	**Speedup**
Cholesky Decomposition	1,058.27	11.440	92.5
Random Normal Gen.	1.64	0.071	23.1
Matrix Multiplication	316.16	3.730	84.8

Speedup and timing data comparing sequential, GPU, and COPULA methods in Figure 3, Figure 4, and Figure 5 were collected on a machine with an Intel Xeon Gold 6126 CPU

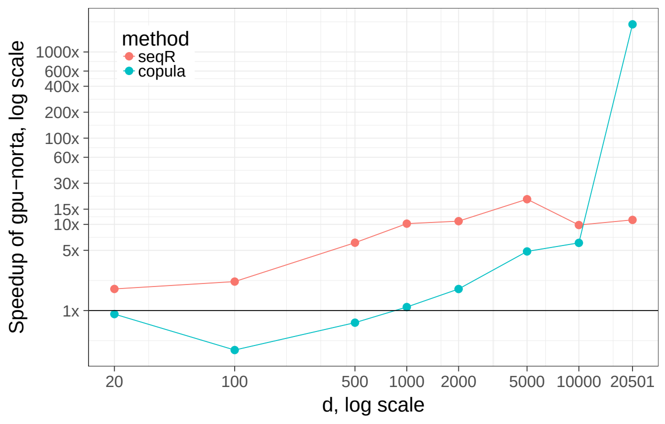

Fig. 3. Speedup of GPU-NORTA vs Sequential NORTA (R version) and GPU-NORTA vs Sequential COPULA (for $n = 2,000$).

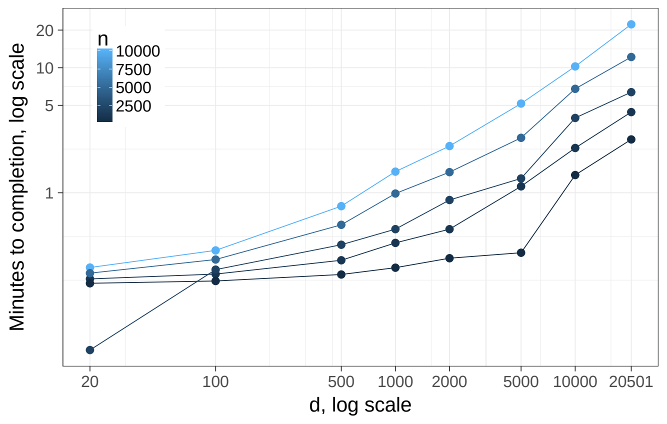

Fig. 5. GPU-NORTA timings with increasing n and d..

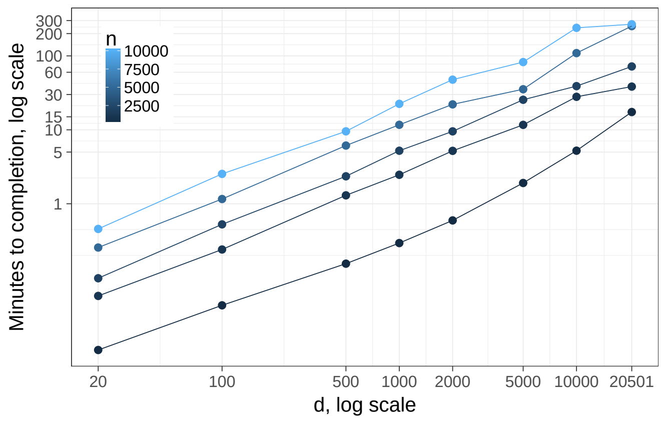

Fig. 4. Sequential NORTA timings (R version) with increasing n and d.

running at 2.60 GHz. The GPU was a Nvidia Tesla P100. Using this machine, and CUDA Unified Memory, we were able to overcome memory limitations on the Macbook Air for larger values of n.

VII. Discussion

Table I summarizes the timings of the functions that were parallelized from sequential R version to GPU CUDA version and provides the speedups. Correlation matrix and distribution file read in timings are not included since they were done sequentially in both cases. There is potential to improve these steps, for instance, by using binary files instead of text files for input data. The inverse transformation step was not included either for reasons discussed in Section V. The other steps: Cholesky decomposition, matrix multiplication, and random number generation were all responsive to parallelization, and are optimized by Nvidia in their `cuSOLVER`, `cuBLAS`, and `cuRAND` libraries. Speedup of the decomposition step resulted in a 92.5x increase. The random normal generation increased by a factor of 23.1 and matrix multiplication increased by a factor of 84.8 over the sequential version.

Figure 3 shows the speedup of GPU-NORTA compared to both the sequential NORTA and sequential COPULA methods. Speedup data was collected using $n = 2,000$ and various d. Speedups were calculated for the whole program running, including data read in and inverse transformation. GPU-NORTA performed faster than sequential NORTA at all d sizes. The speedup peaks (19.6x) at $d = 5,000$ for sequential R comparison and a dramatic increase (2093x) at $d = 20,501$ for comparison against COPULA package. The spike in COPULA is because computation time does not scale linearly.

Figure 4 shows the total run time of the sequential version for various n and d dimensions and Figure 5 shows the total run time for the GPU-NORTA implementation. The GPU-NORTA version took less than 30 min to run for the largest set tested, compared to over 200 minutes for the sequential version. Even with non-optimized read in functions and sequential inverse transformation, GPU-NORTA had notable speedup that increases with increasing data size compared to optimized R code and COPULA packages.

Table II displays the quadratic loss between the simulated correlation matrix $\hat{R}$ from the input (target) correlation, R for $n = 2,000$. Quadratic loss is given by the expression $||\hat{R}R^{-1} - I||_2$ and we compare our proposed GPU-NORTA to the R-NORTA and COPULA implementation. Both our GPU-NORTA and R-NORTA forgo the grid search for an adjusted input matrix, whereas the COPULA approach does not require such an adjustment *a priori*. In our limited studies, at high dimension, we found no substantive loss in accuracy and can safely avoid the search suggested by Cairo and Nelson [7]. Future studies, however, are needed to fully assess the effects.

TABLE II
QUADRATIC LOSSES OF SEQUENTIAL AND GPU IMPLEMENTATION AT VARIOUS d AND USING $n = 2000$

d	1,000	2,000	5,000	10k	20,501
GPU-NORTA	1,275	5,060	7,554	10,225	9,704
R-NORTA	1,232	4,825	7,720	10,571	10,572
COPULA	1,131	4,853	7,524	10,272	10,426

The speedup of the decomposition and matrix multiplications led to a huge gain which helped to off balance the slow run times of sequential read in function and inverse transformation function. There is less significant speedup of the random normal matrix generation, but it still provided a 23.1 times speed.

VIII. Conclusion and Future Work

In this paper, we have introduced a GPU accelerated version of simulating multivarate data with input correlation matrix and arbitrary marginal distributions. This parallel GPU-NORTA algorithm exhibited speedups over a sequential NORTA version and a sequential COPULA version both written in R. The analysis summarized in Table I revealed that the Cholesky and matrix multiplication steps took significant portions of computation time in the sequential version. Fortunately, these functions have attractive parallelization potential and thus had significant speedups over the sequential counterpart. The inverse transformation also has parallelization potiential, but was not implemented in this paper. Future work needs to be done in implementing the statistical functions required for the inverse transformation into CUDA device code. This allows the last step to be parallelized, which should result in more dramatic speedups than what was obtained in this study.

The issue concerning the necessity of adjusting input correlation matrix warrants further study at various dimensions and for other marginal distributions. Our GPU-NORTA implementation could be improved with the ability to allow a user to conduct the search resulting in greater simulation accuracy.

For larger data sets than what was used in this study, this algorithm has the potential to be extended to utilize multiple GPUs. The CUDA API has support for multi-GPU applications [13].

Further speedup could be possible from manual memory management with the trade off of added complexity. The question of whether or not the speedups are worth the complexity trade off needs further exploration. However, results show that even with inefficiencies associated with Unified Memory, the GPU implementation is still faster than both R-NORTA and COPULA versions. With future releases of the CUDA toolkit, Unified Memory management is expected to become more optimized.

Acknowledgment

This material is based upon work supported by the National Science Foundation under grant number IIA1301726. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] B. M. Pucher, O. A. Zeleznik, and G. G. Thallinger, “Comparison and evaluation of integrative methods for the analysis of multilevel omics data: a study based on simulated and experimental cancer data,” *Briefings in Bioinformatics*, pp. 1–11, apr 2018. [Online]. Available: https://academic.oup.com/bib/advance-article/doi/10.1093/bib/bby027/4982568

[2] D. M. Gatti, W. T. Barry, A. B. Nobel, I. Rusyn, and F. A. Wright, “Heading Down the Wrong Pathway: On the Influence of Correlation within Gene Sets,” *BMC Genomics*, vol. 11, no. 1, p. 574, Oct 2010. [Online]. Available: https://doi.org/10.1186/1471-2164-11-574

[3] M. F. Wilkins, C. Morris, and L. Boddy, “A comparison of radial basis function and backpropagation neural networks for identification of marine phytoplankton from multivariate flow cytometry data,” *Bioinformatics*, vol. 10, no. 3, pp. 285–294, Jun 1994. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/10.3.285

[4] T. V. Russkova, “Monte Carlo Simulation of the Solar Radiation Transfer in a Cloudy Atmosphere with the Use of Graphic Processor and NVIDIA CUDA Technology,” *Atmospheric and Oceanic Optics*, vol. 31, no. 2, pp. 119–130, 2018. [Online]. Available: https://link-springer-com.unr.idm.oclc.org/content/pdf/10.1134/S1024856018020100.pdf

[5] K. Häyrinen, K. Saranto, and P. Nykänen, “Definition, structure, content, use and impacts of electronic health records: A review of the research literature,” *International Journal of Medical Informatics*, vol. 77, no. 5, pp. 291–304, may 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1386505607001682

[6] S. T. A. Niaki and B. Abbasi, “Generating Correlation Matrices for Normal Random Vectors in NORTA Algorithm Using Artificial Neural Networks,” *Journal of Uncertain Systems*, vol. 2, no. 3, pp. 192–201, Mar 2008. [Online]. Available: http://www.worldacademicunion.com/journal/jus/jusVol02No3paper04.pdf

[7] M. C. Cario and B. L. Nelson, “Modeling and generating random vectors with arbitrary marginal distributions and correlation matrix,” Northwestern University, Tech. Rep., 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.281

[8] G. Casella and R. L. Berger, *Statistical Inference*. Pacific Grove, CA: Duxbury, 2002.

[9] G. Strang, *Introduction to Linear Algebra*. Wellesley-Cambridge Press, Jan 1993.

[10] M. L. Rizzo, *Statistical Computing with R*. Chapman and Hall/CRC, Nov 2007. [Online]. Available: https://www.taylorfrancis.com/books/9781420010718

[11] C. Genest and J. Mackay, “The joy of copulas: Bivariate distributions with uniform marginals,” *The American Statistician*, vol. 40, no. 4, pp. 280–283, 1986. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/00031305.1986.10475414

[12] M. Sklar, “Fonctions de répartition à n dimensions et leurs marges,” *Publ. inst. statist. univ. Paris*, vol. 8, pp. 229–231, 1959.

[13] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[14] M. S. Nobile, P. Cazzaniga, D. Besozzi, D. Pescini, and G. Mauri, “cuTauLeaping: A GPU-Powered Tau-Leaping Stochastic Simulator for Massive Parallel Analyses of Biological Systems,” *PLoS ONE*, vol. 9, no. 3, p. e91963, Mar 2014. [Online]. Available: http://dx.plos.org/10.1371/journal.pone.0091963

[15] M. Harris, “Unified Memory in CUDA 6,” 2013. [Online]. Available: https://devblogs.nvidia.com/unified-memory-in-cuda-6/

[16] ——, “CUDA 8 Features Revealed: Pascal, Unified Memory and More,” 2016. [Online]. Available: https://devblogs.nvidia.com/cuda-8-features-revealed/

[17] J. Yan, “Enjoy the Joy of Copulas: With a Package copula,” *Journal of Statistical Software*, vol. 21, no. 4, pp. 1–21, Oct 2007. [Online]. Available: http://www.jstatsoft.org/v21/i04/

[18] “cuSOLVER::CUDA Toolkit Documentation,” 2018. [Online]. Available: https://docs.nvidia.com/cuda/cusolver/index.html

[19] “cuRAND::CUDA Toolkit Documentation,” 2018. [Online]. Available: https://docs.nvidia.com/cuda/curand/notices-header.html{\#}notices-header

[20] “cuBLAS::CUDA Toolkit Documentation,” 2018. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[21] K. O’Hara, “StatsLib,” 2018. [Online]. Available: https://github.com/kthohr/stats